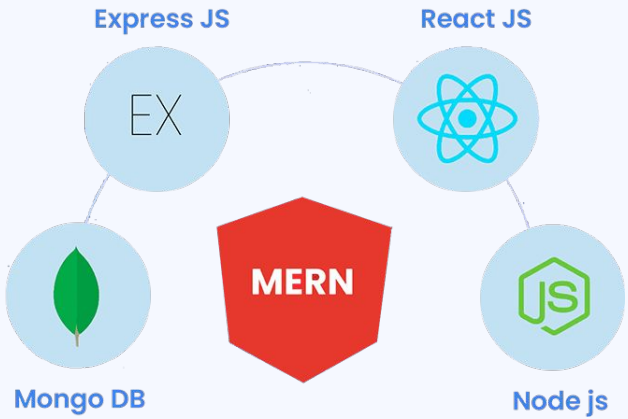


Full Stack Development Using MERN

Day 05 - 25th February 2024

Lecturer Details:

Chanaka Wickramasinghe
cmw@ucsc.cmb.ac.lk
0771570227

Anushka Vithanage
dad@ucsc.cmb.ac.lk
071 527 9016

Prameeth Maduwantha
bap@ucsc.cmb.ac.lk
0772888098

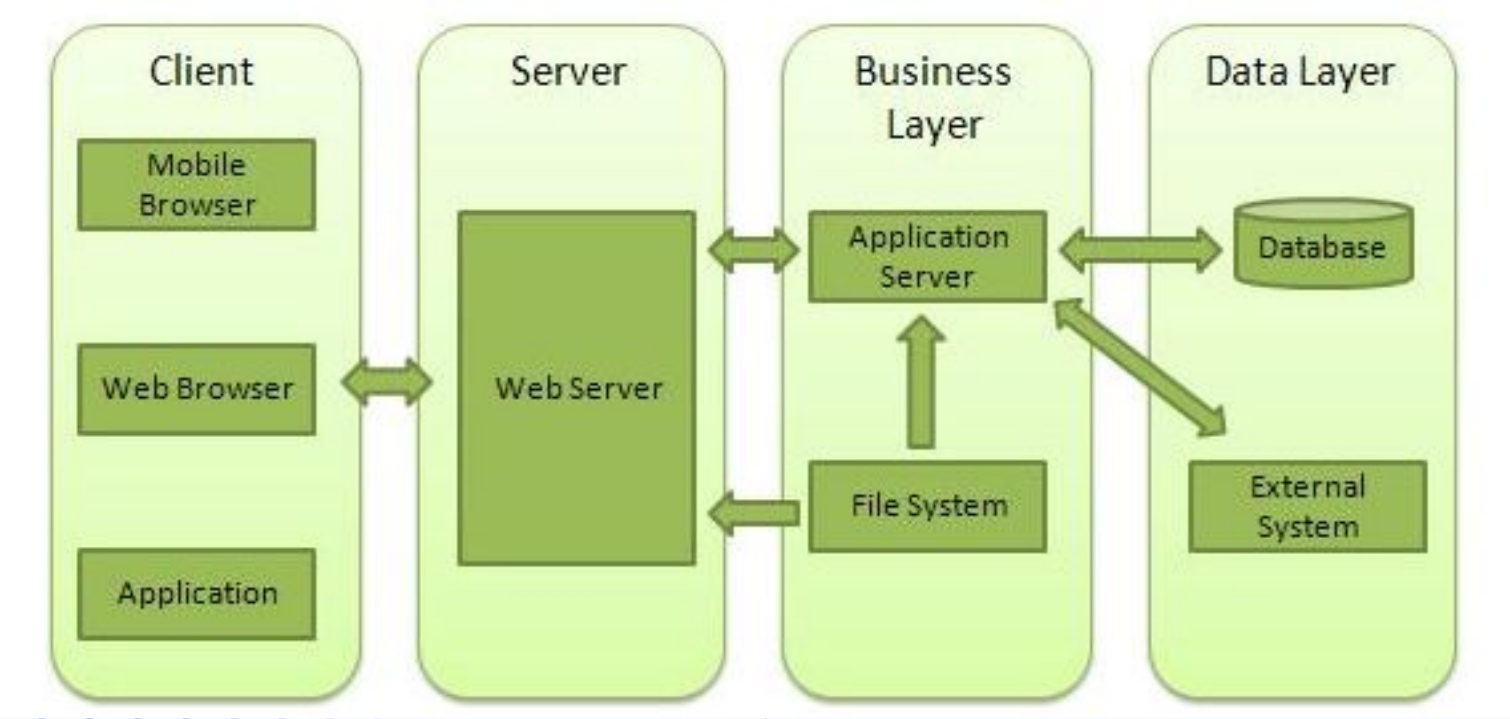
Lecture 05

Building RESTful APIs with Express.js

What is a Web Server?

- A software application which handles HTTP requests from clients (e.g., browsers).
- Returns web pages in response.
- Web servers usually deliver html documents along with images, style sheets, and scripts.
- Web servers usually deliver html documents along with images, style sheets, and scripts.
- Many servers support server-side scripting.
    - Server-side scripting refers to scripts (code) that run on the web server, as opposed to the user's browser (which is client-side scripting).

Web Application Architecture



Web Application Architecture

- Client:
  - Web browsers on computers or mobile devices.
  - They ask (request) for information or pages.
- Server:
  - Receives client's requests and provides responses.
- Business (The Processors):
  - Takes the server's request, figures out the answer, maybe fetches some data, and prepares the reply
- Data:
  - Holds all the detailed info (data) that the Business layer might need to check or store

Express.js

What is Express?

- *A minimal yet flexible* Node.js web application framework.
- Offers *a powerful feature set* for web and mobile application development.
- *Accelerates the development* of Node-based web applications.

Key Features:

- *`Middleware Integration:`* Utilizes middlewares to handle and respond to HTTP requests.
- *`Routing Table:`* Directs HTTP requests based on the method (GET, POST, etc.) and the URL path.
- *`Dynamic HTML:`* Renders HTML pages dynamically by inserting data into templates.

Middleware Integration

What is Middleware?

Middleware in Express refers to *a series of functions that get executed in the order they are added* (or "stacked") whenever a request is made to the server. These functions can access and modify the request and response objects, thus influencing the final output.

Why is it Useful?

Middleware allows developers to *break down the request-handling process into modular chunks*. For example, there might be middleware for logging requests, parsing incoming data, handling authentication, and so on. This makes the code more organized and easier to maintain.

Routing Table

What is Routing?

Routing is the mechanism by which *an application determines how to respond to a client request based on its HTTP method* (like GET, POST) and the URL path. In Express, each route can have one or more handler functions that get executed when the route is matched.

Why is it Useful?

Without routing, your *web application wouldn't know how to respond to a specific request*. For instance, when you visit a website's home page, it's often a GET request to the root ("/") path. If you submit a form, it might be a POST request to a specific endpoint like "/submit-form". Express's routing system makes it simple to define these paths and their behaviors.

Dynamic HTML

What is Dynamic HTML Rendering?

Unlike static HTML pages that remain constant for every user, dynamic HTML pages are generated on-the-fly based on specific conditions or data. In Express, you can integrate with various templating engines (like EJS, Pug, Handlebars) to create HTML content dynamically.

Why is it Useful?

Dynamic rendering allows for *personalized user experiences.* For example, when you log into a web service, your dashboard might display your name, recent activities, and personalized recommendations. This data varies from user to user. By generating HTML dynamically, Express can provide content tailored to individual users based on data from databases, user input, or other sources.

Installing Express

Install the Express framework globally using NPM so that it can be used to create a web application using node terminal.

- **npm install express --save**

You should install the following important modules along with express

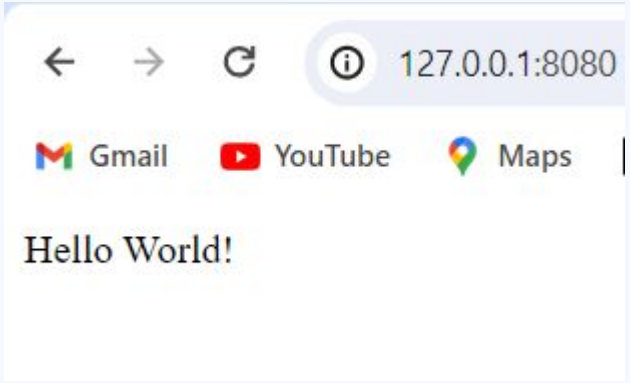
- body-parser - This is a node.js middleware for handling JSON, Raw, Text and URL encoded form data. Eg: Form submission
  - **npm install body-parser --save**
- cookie-parser - Parse Cookie header and populate req.cookies with an object keyed by the cookie names.
  - **npm install cookie-parser --save**

Installing Express Contd

- Multer - This is a node.js middleware for handling multipart/form-data.
  - **npm install multer --save**
- Express-session - Manage user sessions in your Express application.
  - **npm install express-session**
- Cors - Middleware that can be used to enable Cross-Origin Resource Sharing (CORS).
  - **npm install cors**
- Nodemon - Utility that monitors for changes in your node.js app and automatically restarts the server.
  - **npm install nodemon --save-dev**

Express Js - Hello World

```
const express = require('express');

var app = express();

app.get('/', function(req, res){
    res.send('Hello World!');
});

app.listen(8080);
```



Express Js - Hello World

- First, you import the express module by requiring it.
- You create an instance of an Express application by calling express() and assigning it to the variable app.
- app.get(route, callback)
    - This function tells what to do when a get request at the given route is called.
    - The callback function has 2 parameters, request(req) and response(res).
    - The request object(req) represents the HTTP request and has properties for the request query string, parameters, body, HTTP headers, etc. Similarly, the response object represents the HTTP response that the Express app sends when it receives an HTTP request.

Express Js - Hello World

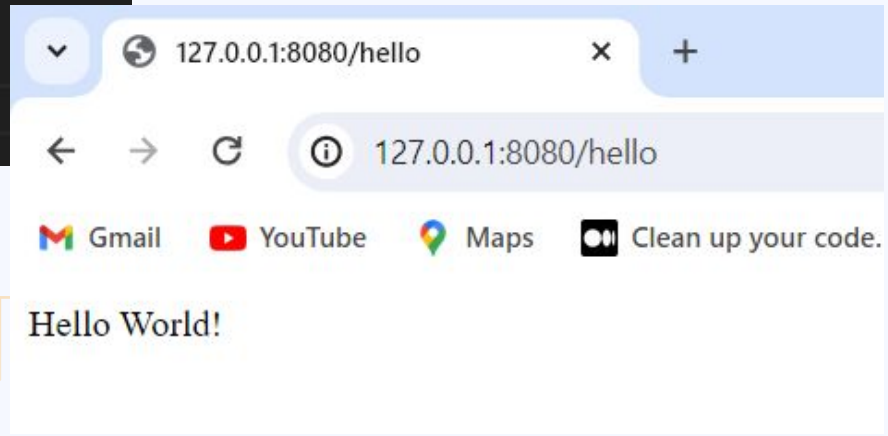
- res.send()
  - This function takes an object as input and it sends this to the requesting client. Here we are sending the string "Hello World!".
- app.listen(port, [host], [backlog], [callback]])
  - This function binds and listens for connections on the specified host and port. Port is the only required parameter here.

Routing

- Web frameworks provide resources such as HTML pages, scripts, images, etc. at different routes.
- To define routes in an Express application
  - **app.method(path, handler)**
- This method can be applied to any of HTTP verbs
  - Get
  - Set
  - Put
  - Delete
- **Path -** Route at which the request will run
- **Handler -** a callback function that executes when a matching request type is found on the relevant route.

Express > JS index.js > ...

```
1  const express = require('express');
2
3  var app = express();
4
5  app.get('/hello', function(req, res){
6      res.send('Hello World!');
7  });
8
9  app.listen(8080);
```



Routers

- Maintaining routes like above is very tedious to maintain.
- To separate the routes from index.js we will use **express.router**

Express > routes.js > ...

```
1  var express = require('express');
2  var router = express.Router();
3
4  router.get('/', function(req, res){
5      res.send('Get route');
6  });
7
8  router.post('/', function(req, res){
9      res.send('Post route');
10 });
11
12 module.exports = router;
```

Express > index.js > ...

```
1  const express = require('express');
2
3  var app = express();
4  var rout = require('./routes.js')
5
6  app.use('/routes',rout);
7
8  app.listen(8080);
9
```

Multiple Methods at same route

```
const express = require('express');

var app = express();

app.get('/hello', function(req, res){
    res.send('Hello World!');
});

app.post('/hello', function(req, res){
    res.send("Pose method called");
});

app.all('/all', function(req, res){
    res.send("All method called");
});

app.listen(8080);
```

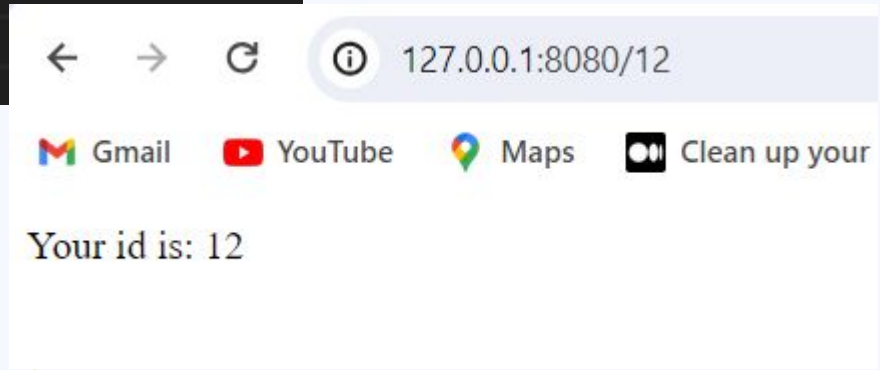
HTTP Methods

S.N.	Method and Description
1	**GET** The GET method is used to retrieve information from the given server using a given URI. Requests using GET should only retrieve data and should have no other effect on the data.
2	**HEAD** Same as GET, but transfers the status line and header section only.
3	**POST** A POST request is used to send data to the server, for example, customer information, file upload, etc. using HTML forms.
4	**PUT** Replaces all current representations of the target resource with the uploaded content.
5	**DELETE** Removes all current representations of the target resource given by a URI.
6	**CONNECT** Establishes a tunnel to the server identified by a given URI.
7	**OPTIONS** Describes the communication options for the target resource.
8	**TRACE** Performs a message loop-back test along the path to the target resource.

Dynamic Routes

Express > JS dynamicroute.js > ...

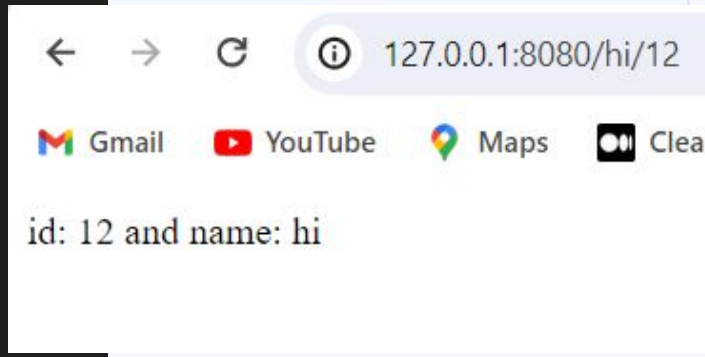
```
1  const express = require('express');
2  var app = express();
3
4  app.get('/:id',function(req, res){
5      res.send('Your id is: ' + req.params.id);
6  });
7
8  app.listen(8080);
```



Dynamic Routes ctd.

Express > JS dynamicroute.js > app.get('/:name/:id') callback

```
1  const express = require('express');
2  var app = express();
3
4  app.get('/:id',function(req, res){
5      res.send('Your id is: ' + req.params.id);
6  });
7
8  app.listen(8080);
9
10 app.get('/:name/:id',function(req, res){
11     res.send('id: ' + req.params.id +' and name: ' + req.params.name)
12 });
13
```



Dynamic Routes Activity

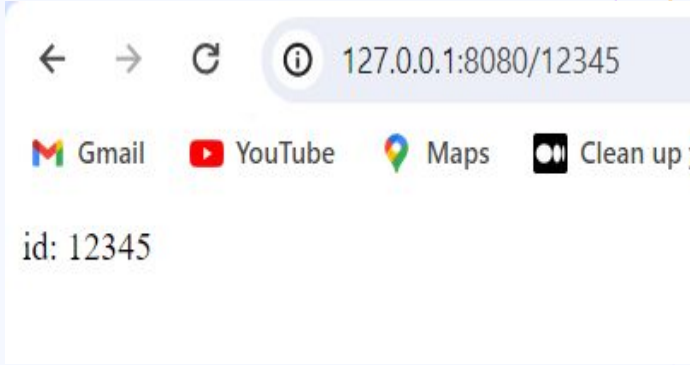
1. Design a route /products/:category/:subCategory?. Note the ? that makes subCategory optional. This route should handle both /products/electronics (return and send "Showing products for category: electronics") and /products/electronics/laptops (return and send "Showing laptops in electronics category").

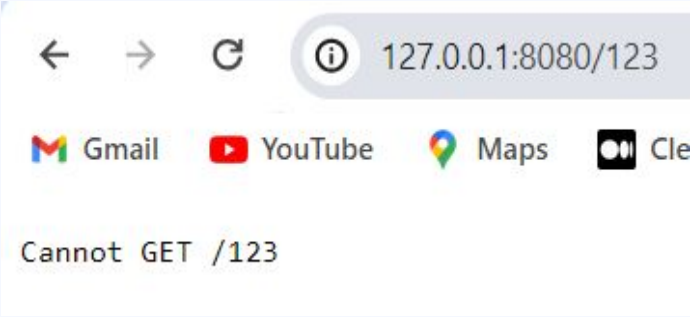
1. Create a dynamic route in an Express.js application to handle requests for user profiles. The route pattern should follow the format /users/:username and should retrieve information about the specified user.

Pattern Matched Routes

Express > JS dynamicroute.js > ...

```
1  const express = require('express');
2  var app = express();
3
4  app.get('/:id([0-9]{5})',function(req, res){
5      res.send('id: ' + req.params.id)
6  });
7
8  app.listen(8080);
9
```





Pattern Matched Routes ctd.

- Above example will only match the requests that have a 5 digit long id.
- More complex regexes can be used to match/validate your routes.

Express > JS dynamicroute.js > ...

```
1   const express = require('express');
2   var app = express();
3
4   app.get('/:id([0-9]{5})',function(req, res){
5       res.send('id: ' + req.params.id)
6   });
7
8
9   /*app.get('/:id',function(req, res){
10      res.send('Your id is: ' + req.params.id);
11  });*/
12
13  app.get('/:name/:id',function(req, res){
14      res.send('id: ' + req.params.id +' and name: ' + req.params.name)
15  });
16
17  app.get('*',function(req, res){
18      res.send('404');
19  });
20
21  app.listen(8080);
22
```

Pattern Matched Routes Activity

1. Construct a route that matches dates in the format YYYY-MM-DD using regex. For example: /events/:date(\\d{4}-\\d{2}-\\d{2}). Respond with the date if matched.

Homework

1. Create a route that matches file requests with specific extensions, e.g., .jpg, .png, or .gif. For instance, /files/:filename(\\w+\\.(jpg|png|gif)). Return the filename when it matches

Node.js - Web Module (Recap)

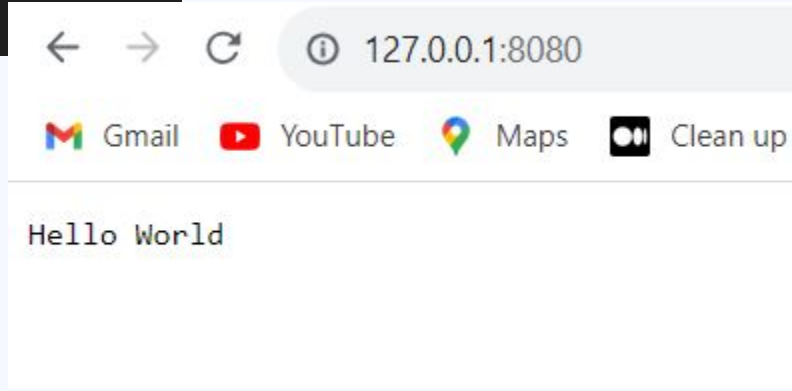
Creating a Web Server using Node

- A web-based Node.js application consists of the following important components:
  - Import required modules
    - Lode Node.js modules using the require directive
  - Create server
    - Create a server to listen the client's requests
  - Read request and return response
    - Read the client request made using browser or console and return the response.

Creating a Web Server Example

firstwebapp.js

```
1  var http = require('http');
2
3  http.createServer(function(req, res){
4      res.writeHead(200, {'Content-Type': 'text/plain'});
5      res.end('Hello World\n');
6  }).listen(8080);
```



Retrieve an HTML file

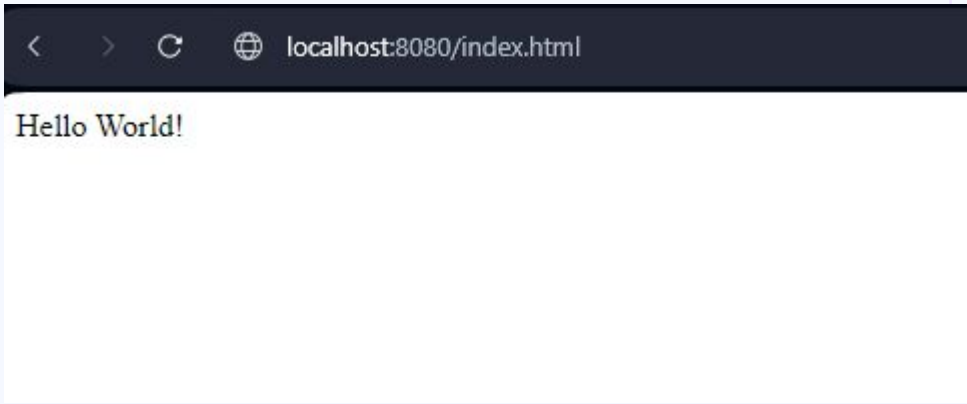
- After the request is received then server extracts the specific path or file the user is trying to access from the request's URL.
- Fetching the File:
  - The server attempts to find and read the requested file from its file system.
  - If the file exists, it's read into memory, ready to be sent back to the user.
  - If the file doesn't exist, an error is noted.
- Responding to the User:
  - If the file is found: The server responds with a "200 OK" status and sends the file to the user.
  - If the file isn't found: The server responds with a "404 Not Found" status, indicating the file is missing.

Retrieve an HTML file

```
var http = require('http');
var fs = require('fs');
var url = require('url');

// Create a server
http.createServer( function (request, response) {
   // Parse the request containing file name
   var pathname = url.parse(request.url).pathname;

   // Print the name of the file for which request is made.
   console.log("Request for " + pathname + " received.");

   // Read the requested file content from file system
   fs.readFile(pathname.substring(1), function(err, data) {
      if (err) {
         console.log(err);
         response.writeHead(404, {'Content-Type': 'text/html'});
      } else {
         //Page found
         response.writeHead(200, {'Content-Type': 'text/html'});
         // Write the content of the file to response body
         response.write(data);
      }

      // Send the response body
      response.end();
   });
}).listen(8080);

// Console will print the message
console.log('Server running at http://127.0.0.1:8080/');
```

Retrieve an HTML file Contd

```
<html>
    <head>
        <title>Sample Page</title>
    </head>

    <body>
        Hello World!
    </body>
</html>
```



Creating Web client using Node

- When the client formulates an HTTP request, targeting a specific server endpoint (localhost:8080/index.html).
- Upon receiving a server response:
  - Data from the server is accumulated in small chunks.
  - This data typically represents the content of a requested webpage or file.
- The accumulated data is saved locally the content is then saved locally as 'temp.html'.
- The client then activates an external tool or module (like open) to display 'temp.html' in the user's default web browser.
- Once the content has been displayed, or if there was an error, the client's current request process completes. However, it remains ready for subsequent requests or actions.

Creating Web client using Example

```
var http = require('http');
var fs = require('fs');

// Options to be used by request
var options = {
    host: 'localhost',
    port: '8080',
    path: '/index.html'
};

// Callback function is used to deal with response
var callback = function(response) {
   // Continuously update stream with data
   var body = '';
   response.on('data', function(data) {
      body += data;
   });

   response.on('end', async function() {
      // Data received completely.
      // Save it to a temporary file
      fs.writeFileSync('temp.html', body);
      // Dynamically import the 'open' module
      const openModule = await import('open');
      const open = openModule.default;

      // Automatically open it in the default web browser
      open('temp.html');
   });
}

// Make a request to the server
var req = http.request(options, callback);
req.end();
```

Creating Web client using Node

```
// Options to be used by request
var options = {
    host: 'localhost',
    port: '8080',
    path: '/index.html'
};
```

- These are the options for the HTTP request. You're specifying that you want to make a request to http://localhost:8080/index.html.

```
// Continuously update stream with data
var body = '';
response.on('data', function(data) {
    body += data;
});
```

- The response object is a stream. As data chunks arrive, they're concatenated to the body string.

Creating Web client using Node

```
response.on('end', async function() {
    // Data received completely.
    // Save it to a temporary file
    fs.writeFileSync('temp.html', body);

    // Dynamically import the 'open' module
    const openModule = await import('open');
    const open = openModule.default;

    // Automatically open it in the default web browser
    open('temp.html');
});
```

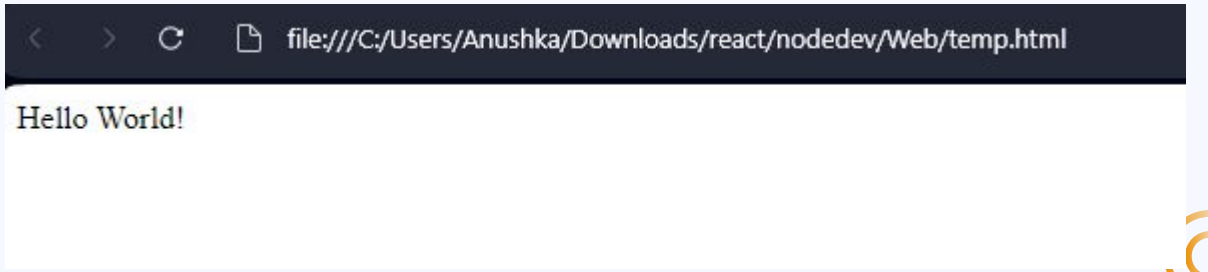
```
// Make a request to the server
var req = http.request(options, callback);
req.end();
```

- Once the entire response has been received, The accumulated body (which contains the content of /index.html fetched from the server) is written to a local file named temp.html
- Open module extracts the default function from the module and uses it to open temp.html in the default web browser.
- **http.request(options, callback)** initiates the HTTP request to the server based on the provided options and uses the callback to handle the response.

Creating Web client using Example Contd

```
node server.js
Server running at http://127.0.0.1:8080/
Request for /index.html received.
Request for /index.html received.
Request for /index.html received.
```

```
node client.js
```

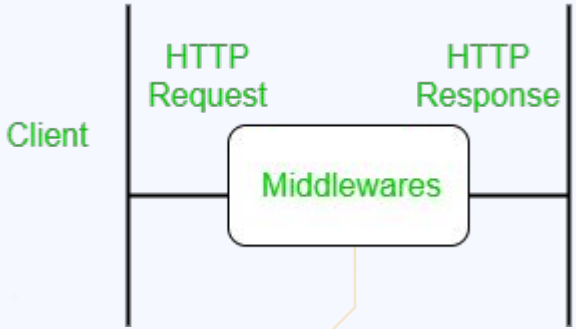


Node.js HTTP Response

- response.writeHead
  - HTTP response header. Headers convey metadata about the HTTP response. They tell the client what type of data to expect, how to cache it, and more.
- response.write(...)
  - Streams a pieces of response body to the client. Useful for sending large files or data in chunks.
- response.end(...)
  - Closes the response stream. You tell the client that you've sent all the data you're going to send.
- response.on(...)
  - Listens for events on the response object, but it's typically used more often on the request object. Tracking when data is being received or when a request/response stream ends.

Middleware

- Middleware functions are functions that have access to the **request object (req)**, t**he response object (res)**, and **the next middleware function** in the application's request-response cycle.
- These functions are used to modify req and res objects for tasks like parsing request bodies, adding response headers, etc.



Express > JS middleware.js > ...

```
1   const express = require('express');
2
3   var app = express();
4
5   app.use(function(req, res, next) {
6       console.log("request recieved at"+Date.now());
7       next();
8   });
9
10  app.listen(8080);
```

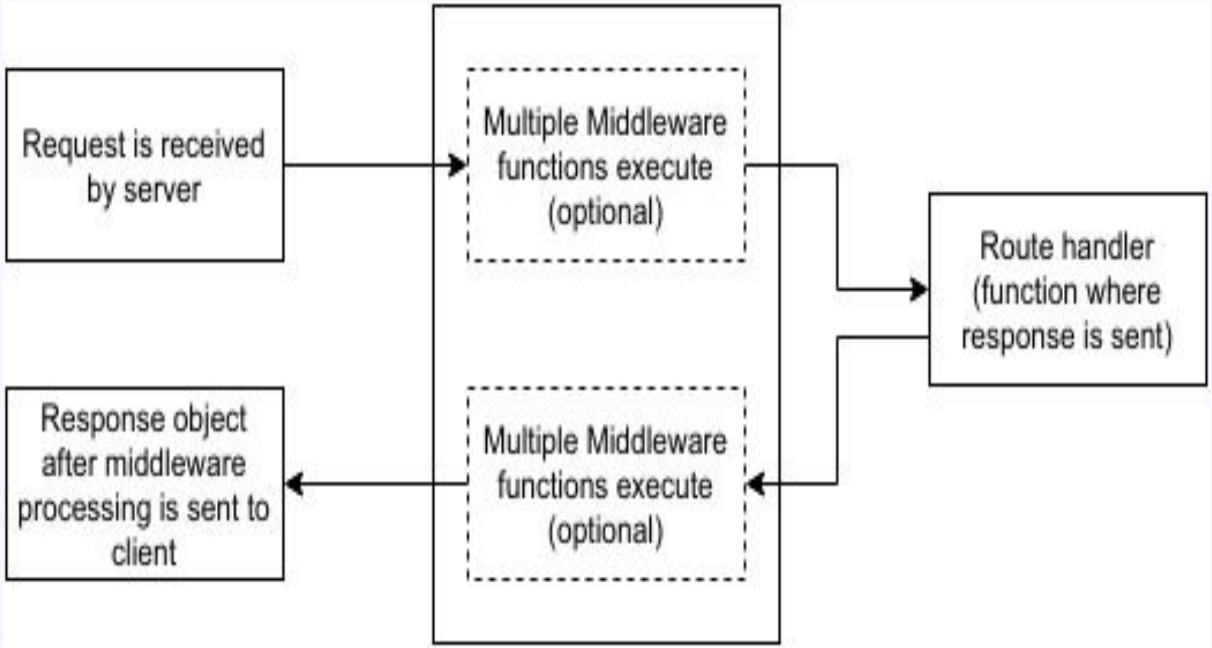
Express > JS middleware.js > ...

```
1   const express = require('express');
2
3   var app = express();
4
5   app.use('/pages',function(req, res, next) {
6       console.log("request recieved at"+Date.now());
7       next();
8   });
9
10  app.get('/pages', function(req, res){
11          res.send('Pages');
12
13  });
14
15  app.listen(8080);
16
```

Order of Middleware calls

Middleware Chaining: Middleware can be chained from one to another, Hence creating a chain of functions that are executed in order. The last function sends the response back to the browser. So, before sending the response back to the browser the different middleware process the request.

The next() function in the express is responsible for calling the next middleware function if there is one.


Request is received by server
Multiple Middleware functions execute (optional)
Route handler (function where response is sent)
Response object after middleware processing is sent to client
Multiple Middleware functions execute (optional)

3rd Party Middleware

- Body-parser
  - This is used to parse the body of requests which have payloads attached to them.
- Cookie - parser
  - It parses Cookie header and populate req.cookies with an object keyed by cookie names.
- Express-session
  - It creates a session middleware with the given options.

Body-Parser

Express > JS bodyParser.js > ...

```
1   const express = require('express');
2   const bodyParser = require('body-parser');
3
4   const app = express();
5
6   // Use body-parser middleware to parse JSON and form data
7   app.use(bodyParser.json());
8   app.use(bodyParser.urlencoded({ extended: true }));
9
10  app.post('/example', (req, res) => {
11    console.log(req.body); // Access the parsed data in req.body
12    res.send('POST request received');
13  });
14
15  app.listen(8080, () => {
16    console.log('Server is running on http://localhost:8080');
17  });
18
```

Cookie-Parser

Express > JS cookieParser.js > ...

```
1  const express = require('express');
2  const cookieParser = require('cookie-parser');
3
4  const app = express();
5
6  // Use cookie-parser middleware to parse cookies
7  app.use(cookieParser());
8
9  app.get('/example', (req, res) => {
10   console.log(req.cookies); // Access the parsed cookies in req.cookies
11   res.send('GET request received');
12 });
13
14 app.listen(8080, () => {
15   console.log('Server is running on http://localhost:8080');
16 });
17
```

Express-Session

Express > JS expressSession.js > app.get('/example') callback

```
1  const express = require('express');
2  const session = require('express-session');
3
4  const app = express();
5
6  // Use express-session middleware to create sessions
7  app.use(session({
8    secret: 'your-secret-key',
9    resave: false,
10   saveUninitialized: true
11 }));
12
13 app.get('/example', (req, res) => {
14   // Access or modify session data in req.session
15   if (req.session.views) {
16     req.session.views++;
17   } else {
18     req.session.views = 1;
19   }
20
21   res.send(`View count: ${req.session.views}`);
22 });
23
24 app.listen(8080, () => {
25   console.log('Server is running on http://localhost:8080');
26 });
27
```

What is an API

- An application programming interface (API) defines the rules that you must follow to communicate with other software systems.
- Developers expose or create APIs so that other applications can communicate with their applications programmatically.
- You can think of a web API as a gateway between clients and resources on the web.

Clients

- Clients are users who want to access information from the web.
- The client can be a person or a software system that uses the API.
- For example, developers can write programs that access weather data from a weather system. Or you can access the same data from your browser when you visit the weather website directly.

Resources

- Resources are the information that different applications provide to their clients.
- Resources can be images, videos, text, numbers, or any type of data.
- The machine that gives the resource to the client is also called the server. Organizations use APIs to share resources and provide web services while maintaining security, control, and authentication.
- In addition, APIs help them to determine which clients get access to specific internal resources.

What is REST

- Representational State Transfer (REST) is a software architecture that imposes conditions on how an API should work.
- API developers can design APIs using several different architectures.
- APIs that follow the REST architectural style are called REST APIs.
- Web services that implement REST architecture are called RESTful web services.

What does the RESTful API client request contain?

- **Unique resource identifier**
  - The URL specifies the path to the resource.
  - The URL is also called the request endpoint and clearly specifies to the server what the client requires.
- **Method**
  - GET
  - POST
  - PUT
  - DELETE
- **HTTP Headers**

What does the RESTful API server response contain?

- **Status Line**

200 – The request was successful. The server has successfully fulfilled the request, and there is a response body that may contain the requested data.

201 – The request has been fulfilled, and a new resource has been created as a result. The newly created resource is returned in the body of the response.

404 – The server could not find the requested resource. This status code indicates that the server did not find the requested URL on the server. It's a client error, suggesting that the client may have mistyped the URL or the resource may no longer be available on the server.

- **Message Body**
    - The response body contains the resource representation.

RESTFul APIs

- **Re**presentational **S**tate **T**ransfer (REST) is an architectural style that defines a set of constraints to be used for creating web services.
- RESTful API is an interface that two computer systems use to exchange information securely over the internet. Most business applications have to communicate with other internal and third-party applications to perform various tasks.

RESTFul APIs ctd.

- **Working**: A request is sent from client to server in the form of a web URL as HTTP **GET** or **POST** or **PUT** or **DELETE** request.
- After that, a response comes back from the server in the form of a resource which can be anything like HTML, XML, Image, or JSON.
- But now JSON is the most popular format being used in Web Services.

Method	URI	Details	Function
GET	/movies	Safe, cachable	Gets the list of all movies and their details
GET	/movies/1234	Safe, cachable	Gets the details of Movie id 1234
POST	/movies	N/A	Creates a new movie with the details provided. Response contains the URI for this newly created resource.
PUT	/movies/1234	Idempotent	Modifies movie id 1234(creates one if it doesn't already exist). Response contains the URI for this newly created resource.
DELETE	/movies/1234	Idempotent	Movie id 1234 should be deleted, if it exists. Response should contain the status of the request.
DELETE or PUT	/movies	Invalid	Should be invalid. **DELETE** and **PUT** should specify which resource they are working on.

Example - Set Up

```
var express = require('express');


var app = express();

//Require the Router we defined in movies.js
var movies = require('./movies.js');

app.use(express.json());
//Use the Router on the sub route /movies
app.use('/movies', movies);

app.listen(3000);
```

Example - Set Up Ctd.

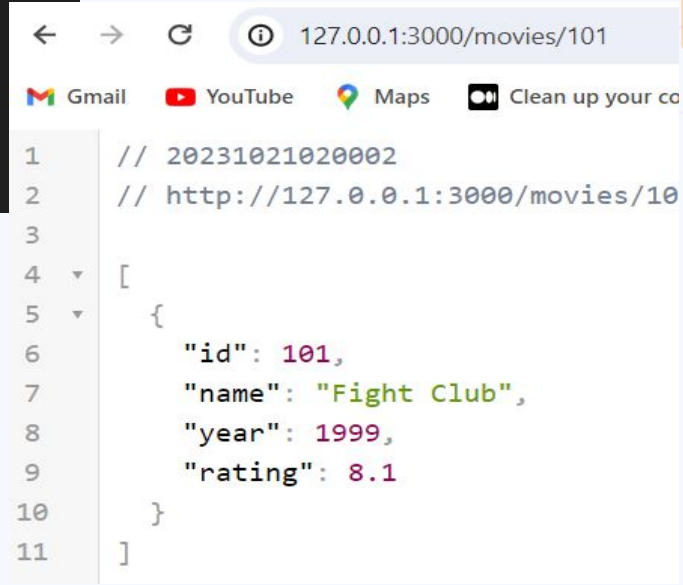
```
var express = require('express');
var router = express.Router();
var movies = [
   {id: 101, name: "Fight Club", year: 1999, rating: 8.1},
   {id: 102, name: "Inception", year: 2010, rating: 8.7},
   {id: 103, name: "The Dark Knight", year: 2008, rating: 9},
   {id: 104, name: "12 Angry Men", year: 1957, rating: 8.9}
];



module.exports = router;
```

GET

```
router.get('/', function(req, res) {
    res.json(movies);
});
```

GET by parameter

```
router.get('/:id', function(req, res) {
    var id = req.params.id;
    var movie = movies.filter(function(movie) {
        return movie.id == id;
    });
    res.json(movie);
});
```



POST

```
router.post('/', function(req, res) {
    movies.push(req.body);
    res.json(req.body);
    console.log(req.body);
});
```

```
[nodemon] starting `node index.js`
{ id: 105, name: 'Pulp Fiction', year: 1994, rating: 8.9 }
```

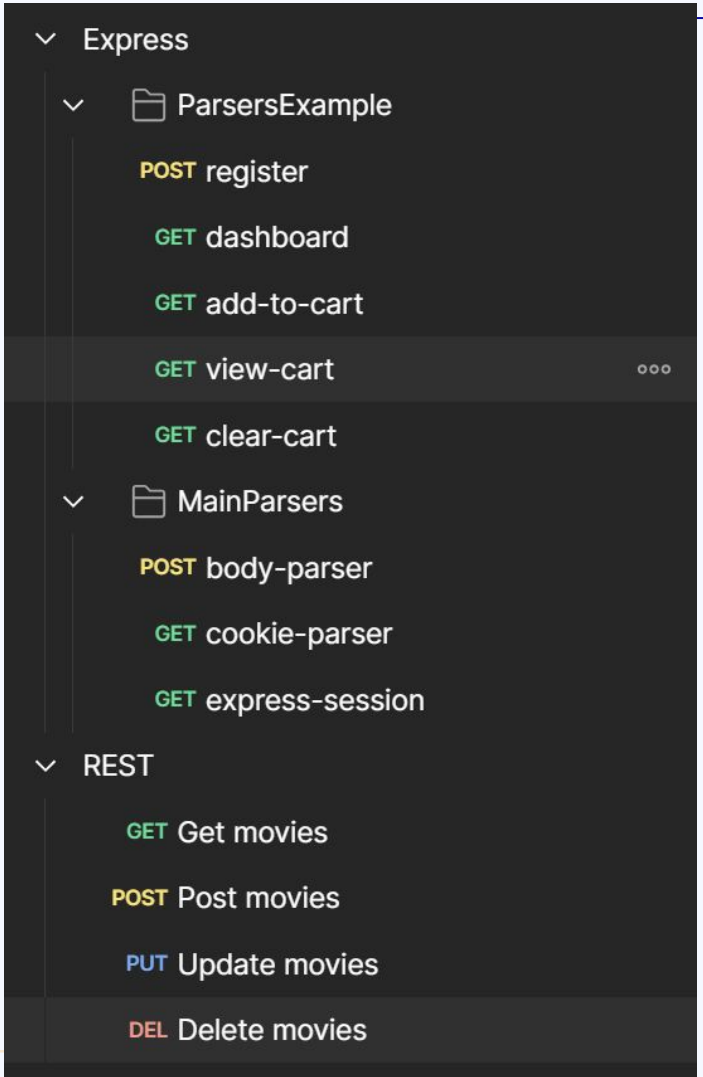
PUT

```
router.put('/:id', function(req, res) {
    var id = req.params.id;

    // Find the index of the movie with the given ID
    var index = movies.findIndex(function(movie) {
        return movie.id == id;
    });

    if (index === -1) {
        // Movie with the given ID does not exist
        return res.status(404).json({ error: 'Movie not found' });
    }

    // Update the movie at the found index with the new data
    movies[index] = { ...movies[index], ...req.body };

    res.json(movies[index]);
});
```

DELETE

```
router.delete('/:id', function(req, res) {
    var id = req.params.id;
    // Find the index of the movie with the given ID
    var index = movies.findIndex(function(movie) {
        return movie.id == id;
    });

    if (index === -1) {
        // Movie with the given ID does not exist
        return res.status(404).json({ error: 'Movie not found' });
    }

    movies.splice(index, 1);
    res.json(movies);
    console.log(movies);
});
```

Postman


Express
ParsersExample
POST register
GET dashboard
GET add-to-cart
GET view-cart
GET clear-cart
MainParsers
POST body-parser
GET cookie-parser
GET express-session
REST
GET Get movies
POST Post movies
PUT Update movies
DEL Delete movies

Integration of backend and frontend

- Middleware such as body-parser to handle incoming data.
- Ensure both frontend and backend are running on different ports.
- Use CORS (Cross-Origin Resource Sharing) middleware in the backend to handle cross-origin requests.
    - Security feature implemented by web browsers that controls how web pages in one domain can make requests to another domain.
- Axios in the frontend to send HTTP requests.

Axios

Axios is a popular JavaScript library that is used to make HTTP requests from a web browser or a Node.js application. It provides a simple and convenient API for performing asynchronous operations with HTTP.

Install Axios

```
npm install axios
```

Setting up the environment

1. Create a new project
2. Create react app for client
3. Install axios for client
4. Create a separate folder for server
5. Initialize the server (npm init)
6. Install express, cors, body-parser (npm install _____)

Example - Client

```jsx
import React, { useState } from 'react';
import axios from 'axios';

const MessageSender = () => {
    const [message, setMessage] = useState('');
    const [response, setResponse] = useState('');

    const sendToServer = () => {
        axios.post('http://localhost:3001/change-message', { message })
            .then(res => setResponse(res.data.message))
            .catch(() => setResponse('Error sending to server'));
    };
    return (
        <div>
            <input
                type="text"
                value={message}
                onChange={e => setMessage(e.target.value)}
            />
            <button onClick={sendToServer}>Send</button>
            <div>{response}</div>
        </div>
    );
};

export default MessageSender;
```

Example contd- Sending data to server

```
const sendToServer = () => {
    axios.post('http://localhost:3001/change-message', { message })
        .then(res => setResponse(res.data.message))
        .catch(() => setResponse('Error sending to server'));
};
```

- Here, axios.post is used to send a POST request. It has two main arguments:
    - The first argument is the URL to which the request is being sent. In this case, it's '**http://localhost:3001/change-message**'.
    - The second argument is the data being sent as the request body. Here, an object **{ message }** is being sent. This object contains the message which the function likely fetches from the component's state or props.

Example contd- Sending data to server

```
const sendToServer = () => {
    axios.post('http://localhost:3001/change-message', { message })
        .then(res => setResponse(res.data.message))
        .catch(() => setResponse('Error sending to server'));
};
```

- **then()** method handles the response from the server if the request was successful. res is the response object received from the server.
- **setResponse(res.data.message)** assumes that the server returns a JSON object with a property message, and this message is set to the component's state using the setResponse function.
- **catch()** method catches any errors that occur during the request

Example - Server

MessegeSender.jsx | App.js | index.js | routes.js

server > index.js > ...

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3  const cors = require('cors');
4  const routes = require('./routes'); // Import routes
5
6  const app = express();
7  const PORT = 3001;
8
9  app.use(cors());
10 app.use(bodyParser.json());
11
12 app.use('/', routes); // Mount routes at the root path
13
14 app.listen(PORT, () => {
15   console.log(`Server is running on http://localhost:${PORT}`);
16 });
17
```

Example - Server

MessegeSender.jsx | App.js | index.js | routes.js

server > routes.js > ...

```
1   const express = require('express');
2   const router = express.Router();
3
4   router.post('/change-message', (req, res) => {
5     const clientMessage = req.body.message;
6     const changedMessage = `Server says: ${clientMessage}`;
7     res.json({ message: changedMessage });
8   });
9
10  module.exports = router; // Export the router
11
12
```

Example contd- Receiving from server

```
app.post('/change-message', (req, res) => {
    const clientMessage = req.body.message;
    const changedMessage = `Server says: ${clientMessage}`;
    res.json({ message: changedMessage });
});
```

- **app.post(...)** sets up an HTTP POST endpoint on the Express server.
- **'/change-message'** is the path or route on which this endpoint is available. When combined with the domain and port (e.g., http://localhost:3001), the full URL would be **http://localhost:3001/change-message**
- **(req, res) => { ... }** is the callback function that's executed when a POST request is made to the /change-message route.

Example contd- Receiving from server

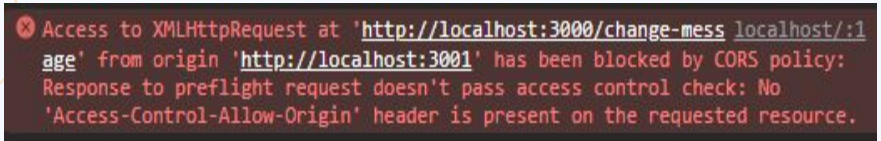
```
app.post('/change-message', (req, res) => {
    const clientMessage = req.body.message;
    const changedMessage = `Server says: ${clientMessage}`;
    res.json({ message: changedMessage });
});
```

- **req** stands for request, contains all the information about the incoming request, including headers, parameters, and the body of the request.
- **res** stands for response. It's an object that allows you to formulate and send back a response to the client.
- **req.body** contains the parsed body of the request, and **.message** accesses the message property of that body. It assumes the client has sent a JSON object that looks like: { "message": "some text" }.
- **res.json({ message: changedMessage })** sends a response back to the client in the form of a JSON object.

Without Cors

- Let's say your server is running on **http://localhost:5000** and your frontend React app is running on **http://localhost:3000**.
- When you make a request from the frontend (on port 3000) to the backend (on port 5000), it's considered a cross-origin request because the port numbers differ.
- Cors allows Cross-Origin Resource Sharing

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const PORT = 5000;

app.use(bodyParser.json());

app.post('/change-message', (req, res) => {
    const clientMessage = req.body.message;
    const changedMessage = `Server says: ${clientMessage}`;
    res.json({ message: changedMessage });
});

app.listen(PORT, () => {
    console.log(`Server is running on http://localhost:${PORT}`);
});
```



Thanks!